Perquirere, 12 (2): 142-157, dez. 2015

http://perquirere.unipam.edu.br

Desenvolvimento de uma ferramenta de apoio à automação de testes de sistema

Development of a tool to support system test automation

Eduardo Henrique Silva
Pós-graduando em Engenharia de Software pelo Centro Universitário de Patos de Minas – UNIPAM.
E-mail: eduardohs@unipam.edu.br

Fernando Corrêa de Mello Junior
Mestre em Engenharia Elétrica e Docente do Centro Universitário de Patos de Minas – UNIPAM.
E-mail: fernandocmjr@unipam.edu.br

Resumo: Empresas de desenvolvimento de software buscam, por meio de ferramentas, aumentarem a produtividade. A tarefa de testes de software é uma atividade que se despende de muitas horas no processo de desenvolvimento. Baseado nas dificuldades em se criar testes, foi desenvolvida uma ferramenta que permite a geração de *scripts* de testes automatizados, que testará as funcionalidades dos sistemas a fim de detectar falhas. Para analisar se a ferramenta foi efetiva, realizou-se uma comparação entre a utilização de um teste manual e um teste automatizado gerado pela ferramenta. A partir dessa análise, foi possível mensurar se houve um ganho de produtividade na etapa de teste e se os testes gerados podem ajudar na detecção de erros.
Palavras-chave: Qualidade de software. Testes de sistema. Testes automatizados.

Abstract: Software development companies are seeking, by means of tools, increase productivity. The software testing task is an activity that spends many hours in the development process. Based on the difficulties in creating tests, a tool has been developed to allow the generation of automated test scripts that will test the functionalities of the systems to detect failures. To analyze if the tool was effective, there was a comparison between the use of manual test and automated test pattern generated by the tool. From this analysis, it was possible to measure if there was a productivity gain in test step and if the generated test can help in error detection.
Keywords: Software quality. System tests. Automated tests.

---

1 INTRODUÇÃO

Atualmente, empresas buscam cada vez mais utilizar sistemas informatizados, com o intuito de gerenciar as suas informações e agilizar os seus processos. A utilização de softwares é um diferencial competitivo entre as empresas, pois permite fluir de maneira hábil informações estratégicas de seus clientes e fornecedores.

Muitas empresas adquirem seus sistemas em fábricas de desenvolvimento de softwares. As fábricas, geralmente, criam novos softwares ou customizam algumas funcionalidades de sistemas prontos para se adaptarem aos processos empresariais de determinadas organizações. A tecnologia de informação vive em constante mudança e precisa de empresas que realizam manutenções evolutivas constantemente nos softwares.

Com a grande demanda de desenvolvimento de sistemas, é imprescindível que haja qualidade no processo de criação e no produto final de um software, para que não exista estouro no tempo de entrega, não ultrapasse o orçamento estabelecido e seja entregue de acordo com os requisitos especificados pelo cliente. Para isso, as empresas de software buscam metodologias e ferramentas que possam apoiá-las no ciclo de desenvolvimento de um software.

O teste de software é uma atividade que permite minimizar o risco de defeitos após a implantação do sistema, contribuindo para uma melhor qualidade final do produto. Muitas vezes, a atividade de teste manual consome muito tempo no processo de desenvolvimento, então uma alternativa é a realização de testes de softwares automatizados. A automação de testes é realizada a partir da escrita de *scripts* por meio de ferramentas ou *frameworks* que são executados por um software de computador. O desenvolvimento de *scripts* de testes pode, ainda, consumir um bom tempo para os desenvolvedores.

Nesse contexto, foi desenvolvida uma ferramenta que permite apoiar os testes de software, a partir da geração de *scripts*, utilizando os frameworks *JUnit* e *Selenium*, com o objetivo de agilizar a atividade de testes. A ferramenta visa acessar a página *web* e capturar os elementos HTML (*HyperText Markup Language*). A partir deles, o testador selecionará uma lista de testes predefinidos para serem gerados automaticamente.

2 REFERENCIAL TEÓRICO

Nesta seção, são abordadas as áreas temáticas que caracterizaram a ferramenta no contexto de testes de software.

2.1 QUALIDADE DE SOFTWARE

De acordo com Oakland (1994), o termo qualidade é subjetivo, podendo variar de acordo com a percepção de cada usuário. A qualidade pode ser considerada como o atendimento das necessidades de um cliente, ou seja, um produto é de qualidade se ele atender às exigências do usuário.

A qualidade de software tem como objetivo garantir que especificações explícitas e necessidades implícitas estejam no produto final. Para garantir a qualidade, normalmente, são utilizadas metodologias de desenvolvimento, ferramentas de apoio e normalização dos processos de desenvolvimento. Atualmente, existem vários modelos que garantem a qualidade do processo, porém, o principal objetivo é garantir a qualidade do produto final para que satisfaça as expectativas do cliente (GUERRA; COLOMBO, 2009).

Para Sommerville (2011), a qualidade do produto está diretamente ligada à qualidade do processo de desenvolvimento. Trabalhando com os processos adequadamente, as chances são maiores de se conduzir a produção do software com uma melhor qualidade. O Quadro 1 apresenta doze atributos de qualidade que estão relacionados ao produto e ao processo.

Quadro 1 - Atributos de qualidade de software

Segurança	Compreensibilidade	Portabilidade
Proteção	Testabilidade	Usabilidade
Confiabilidade	Adaptabilidade	Reusabilidade
Resiliência	Modularidade	Eficiência
Robustez	Complexibilidade	Capacidade de aprendizado

Fonte: Sommerville (2011)

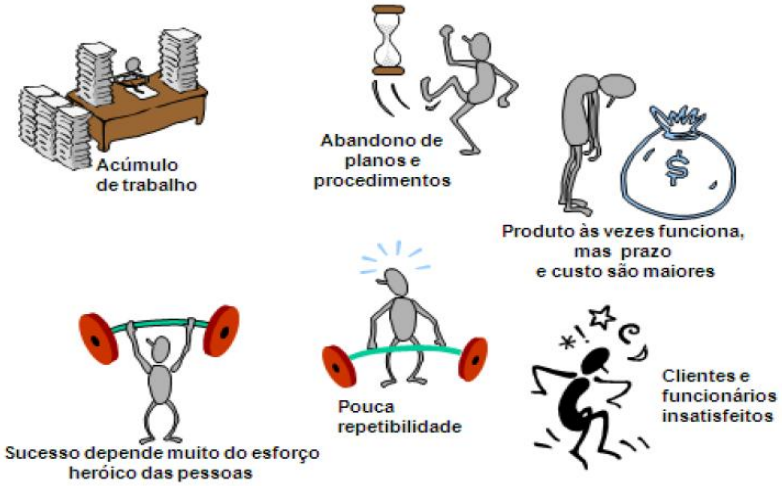
Esses atributos estão relacionados com a confiança, a usabilidade, a eficiência e a manutenibilidade do software. É impossível um sistema otimizar todos esses atributos, mas as empresas devem medir a qualidade do produto de acordo com o cliente e alterar o processo até atingir o nível de qualidade desejado.

2.1.1 Qualidade do processo

O processo de software é a sequência de atividades para desenvolver ou realizar a manutenção em um sistema, utilizando-se metodologias, ferramentas de apoio ao desenvolvimento e pessoas para a execução de tarefas (HUMPHREY, 1995).

A Figura 1 apresenta a situação de muitas organizações de software, na qual existe um grande acúmulo de trabalho, ocasionando o abandono de planos e procedimentos. Como consequência, o produto do software pode funcionar, mas com tempo e custos maiores do que o previsto, acarretando em clientes insatisfeitos (GUERRA; COLOMBO, 2009).

Figura 1- Situação de muitas organizações de software



Fonte: Magnani (1998).

Para Lucinda (2010), a maioria das empresas trabalha visando produzir resultados satisfatórios. Esses resultados são produto da execução de várias atividades ou processos. Para a melhoria dos processos, as organizações devem, primeiramente, aumentar a sua eficiência e eficácia, realizar a distribuição de tarefas entre as equipes e trabalhar em cima de atividades realmente necessárias.

Segundo Sommerville (2011), as empresas devem definir padrões de processo para serem adotados durante o desenvolvimento de software. Nesses padrões, podem ser incluídas definições de especificações, projeto e processos de validação e ferramentas de suporte. Esses padrões não podem ser caros em termos de tempo e esforço ao serem aplicados e também devem gerar melhorias de qualidade no processo.

A ferramenta desenvolvida visa apoiar o processo de testes, pois ela automatiza e padroniza alguns testes de sistema, que antes poderiam ser realizados manualmente ou por meio do desenvolvimento de *scripts* de testes. Portanto, com a utilização da ferramenta a atividade de testes torna-se mais ágil e produz um resultado padronizado de melhor qualidade.

2.1.2 Qualidade do produto

Segundo Rocha, Maldonado e Weber (2001), a qualidade do produto pode ser avaliada a partir de um grupo de características, que podem ser divididas, transformando-se em um grupo de atributos que descrevem a qualidade de um produto. Nesse contexto, é necessário um modelo que ajude a organizar esses atributos e a avaliar a qualidade de um software, determinando se o produto satisfaz ou não as necessidades do cliente.

Para Guerra e Colombo (2009), a satisfação do cliente com o software está diretamente ligada com o seu desempenho e com a ausência de defeitos, falhas ou erros. Sendo assim, a qualidade do produto é alcançada se as necessidades forem cumpridas e se o produto se comporta como o esperado.

A correção de falhas após a implantação do sistema gera um alto custo para as empresas de software, por isso a atividade de teste deve ser bem planejada e cuidadosa. A atividade de testes é um elemento crítico da garantia de qualidade do produto, e não é incomum organizações de software gastarem 40% do esforço total de um projeto em testes (PRESSMAN, 1995).

A ferramenta desenvolvida pretende contribuir com a qualidade do produto, pois ela poderá gerar uma bateria de testes para a funcionalidade selecionada, tentando encontrar o máximo de falhas possíveis.

2.2 TESTES DE SOFTWARE

Segundo Pressman (1995), a atividade de teste tem por objetivo executar um software com a intenção de descobrir erros. Um teste bem-sucedido é aquele que revela uma falha ainda não descoberta.

2.2.1 Casos de teste

Projetar casos de teste é uma atividade desafiadora, pois, muitas vezes, são desenvolvidos testes que podem parecer certos, mas que apresentam pouca garantia de estarem completos. Atualmente, existem métodos de projeto de casos de testes que oferecem ao desenvolvedor uma abordagem sistemática do software. Esses métodos oferecem uma estrutura que ajuda a garantir a integridade do teste, possibilitando uma alta chance de encontrar falhas (PRESSMAN, 1995).

O teste de caixa branca ou teste estrutural é um método de projeto de casos de teste que utiliza a estrutura de controle do projeto procedimental para derivar casos de teste. Esse teste examina a estrutura lógica interna do software, fornecendo casos de teste que põem à prova conjuntos específicos de condições e laços (PRESSMAN, 1995).

Um teste de caixa branca efetuado muito cuidadosamente levaria a uma porcentagem alta na detecção de erros, porém testes exaustivos apresentam certos problemas, tais como, se fosse avaliada uma lógica de programação muito complexa, poderiam ser gastas muitas horas de desenvolvimento da empresa, estourando o planejamento (PRESSMAN, 1995).

Outro método de projeto de casos de teste é o de caixa preta ou teste funcional. Esse método concentra apenas nos requisitos funcionais do software, possibilitando que o desenvolvedor derive um conjunto de entradas que exercitem todos os requisitos funcionais de um software. O teste caixa preta não é uma alternativa ao teste de caixa branca, ele é uma abordagem complementar que permite descobrir erros diferentes dos métodos de caixa branca (PRESSMAN, 1995).

O teste caixa preta, normalmente, é desenvolvido durante as últimas etapas da atividade de teste, pois ele desconsidera as estruturas de controle e concentra no domínio da informação, com o objetivo de validar os requisitos funcionais, o intervalo de dados que o sistema aceita e o efeito de combinações específicas de dados sobre um requisito do software (PRESSMAN, 1995).

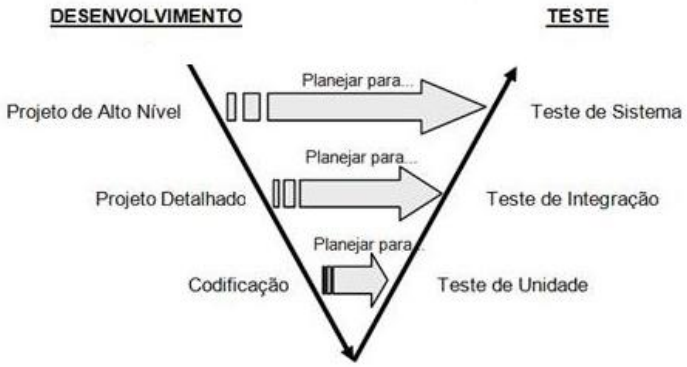
Os métodos de caixa branca e caixa preta podem ser utilizados em conjunto para oferecer uma abordagem que valide a interface do software e garanta o correto funcionamento interno do software (SOMMERVILLE, 2011).

A ferramenta desenvolvida aborda apenas o método de teste de caixa preta, pois ela atua sobre as funcionalidades do software, permitindo que ele seja testado a partir de sua interface *web*. Para a realização do teste, a ferramenta utiliza uma combinação de valores com intuito de agilizar o teste e também permite que o usuário insira suas próprias combinações.

2.2.2 Níveis de testes de desenvolvimento

Os testes de desenvolvimento são todas as atividades que são realizadas pela equipe de desenvolvimento do software. Durante o desenvolvimento, o teste pode ocorrer em três níveis de granularidade (SOMMERVILLE, 2011). A Figura 2 apresenta os três níveis de testes de desenvolvimento em paralelo com o desenvolvimento do software.

Figura 2 - Níveis de teste



Fonte: Adaptado de Craig e Jaskiel (2002)

O planejamento dos testes deve ocorrer em uma abordagem *top-down*, ou seja, primeiramente, deve ocorrer o planejamento dos testes de sistemas, posteriormente, o planejamento dos testes detalhado e, por último, o planejamento dos testes a partir da codificação. Já a execução dos testes deve ocorrer em uma abordagem inversa, conhecida como *bottom-up*. Inicialmente, são realizados os testes a partir da codificação e, por fim, os testes a partir do projeto de alto nível (ROCHA *et al.*, 2001).

Os testes de unidade têm como objetivo testar as menores partes de um software individualmente, como métodos ou classes de objetos. Quando o desenvolvedor está testando um método ou uma classe de objetos, ele deve projetar os testes para fornecer uma cobertura de todas as características do objeto, ou seja, deve testar todas as operações associadas ao objeto, valores de todos os atributos associados ao objeto e colocar o objeto em todos os estados possíveis (SOMMERVILLE, 2011). Segundo Pressman (1995), os testes de unidade baseiam-se sempre na caixa branca, e esse passo pode ser realizado em paralelo para múltiplas unidades.

O teste de integração é uma técnica sistemática para a construção da estrutura do programa, realizando-se, ao mesmo tempo, testes para descobrir erros associados nas interfaces entre os módulos quando esses são integrados para construir a estrutura de programa que foi determinada pelo projeto (PRESSMAN, 1995).

Segundo Sommerville (2011), o teste de sistema envolve a integração de componentes para a criação de uma versão do sistema que será testada. O teste de sistema verifica se os componentes são compatíveis, se eles integram corretamente e se eles transferem os dados corretamente. Para Pressmam (1995), o teste de sistema é uma série de diferentes testes, cujo propósito é pôr à prova uma versão funcional do sistema. Rocha, Maldonado e Weber (2001) definem o teste de sistema como uma simulação do usuário final em busca de erros a partir da utilização, pois os testes devem ser executados no mesmo ambiente, com as mesmas condições e com os mesmos dados que o usuário utiliza diariamente.

A ferramenta desenvolvida apoia a atividade de testes em nível de sistema, pois ela simula a execução do software como um todo, nas mesmas condições que o usuário utiliza o sistema. O desenvolvedor poderá criar testes de sistemas para diferentes linguagens de programação desde que a interface utilize a tecnologia HTML, devido à

ferramenta atuar com testes de alto nível. Se a ferramenta atuasse com testes de unidade, ela teria que ser desenvolvida focada apenas em uma linguagem de programação, pois atuaria diretamente com o código fonte.

2.3 TESTES AUTOMATIZADOS

Para Bernardo (2011), os testes automatizados são programas ou *scripts* que executam as funcionalidades do software em teste e fazem verificações automáticas nos efeitos colaterais obtidos. A utilização do computador para realizar os testes traz benefícios ao projeto, como velocidade de execução de um teste, execução paralela dos testes, facilidade na criação de casos de testes complexos e facilidade em repetir o teste a qualquer momento.

As ferramentas de apoio aos testes automatizados podem reduzir o tempo de realização do teste sem reduzir a eficácia. Atualmente, possui uma série de categorias de ferramentas de testes. A seguir estão relacionadas algumas dessas categorias:

- Analisadores estáticos: permitem realizar uma varredura no código fonte, a fim de encontrar anomalias;
- Auditores de código: permitem verificar a qualidade do software, a fim de garantir que ele atenda a padrões de codificação;
- Geradores de arquivos de teste: permitem a geração de valores previamente determinados para programas que estão em teste;
- Verificadores de teste: permitem a medição da cobertura interna dos testes, a fim de relatar o valor ao gestor de qualidade de software;
- Comparadores de saída: permitem comparar um conjunto de saídas de um programa com outro conjunto para determinar a diferença entre eles;
- Simuladores de ambiente: permitem a modelagem do ambiente externo ao software de tempo real e, depois, simulam dinamicamente as condições operacionais reais.

A ferramenta desenvolvida engloba apenas uma categoria de ferramentas de automatização. Ela é um simulador de ambiente, pois permite que o teste seja realizado simulando o ambiente real de utilização. A ferramenta também utiliza um software de geração de arquivos de teste, que serão utilizados para execução dos *scripts*.

2.3.1 *JUnit*

O *JUnit* é um *framework* para desenvolvimento de testes de unidade em Java. Esse modelo de *framework* é conhecido com *xUnit* e está disponível para diversas linguagens de programação, como ASP, C++, C#, PHP dentre outros. O *JUnit* é uma ferramenta poderosa para testes unitários, porém, devido a sua diversidade, ela também pode ser utilizada para automação de outros tipos de testes de software (MASSOL; HUSTED, 2009).

2.3.2 *Selenium*

O *Selenium* é um *framework* de testes que permite a criação de *scripts* de testes para interfaces web que utilizam o HTML. Por meio dele é possível o desenvolvedor automatizar a execução de funcionalidades em um *browser*. O *Selenium* disponibiliza algumas ferramentas para automação dos testes (BURNS, 2012):

- *Selenium IDE*: é um *plugin* do navegador Firefox que permite a realização e automação de testes. Ele também possibilita a gravação dos testes para o desenvolvedor acompanhar o fluxo de trabalho;
- *Selenium WebDriver*: é um conjunto de bibliotecas que permite a criação de *scripts* de testes em Java para a automação de testes no navegador de internet.

A ferramenta desenvolvida gera *scripts* de testes com o *JUnit* para automatizar os testes e com o *Selenium WebDriver* para realizar a simulação de utilização das funcionalidades do sistema. A ferramenta não realiza os testes diretamente no sistema, apenas oferece um suporte para o desenvolvedor automatizar a geração de alguns *scripts* de testes na linguagem Java. A execução dos *scripts* de testes fica a cargo do desenvolvedor.

3 METODOLOGIA

Este estudo iniciou com um estudo bibliográfico sobre a importância da realização de testes no desenvolvimento de software, de como agilizar a atividade de testes e os principais tipos e metodologias de testes realizados em softwares. A partir desse estudo, foi definida a finalidade da ferramenta, ou seja, ela atuará em testes de sistema, utilizando a metodologia de caixa preta e podendo realizar testes independente de linguagem de programação, desde que a interface seja desenvolvida em HTML. Posteriormente, foram definidos os requisitos principais da ferramenta:

- Captura de elementos das páginas web;
- Apresentação de *checklist* (lista de testes predefinidos gerados pela ferramenta) para a geração de *scripts* de teste;
- Utilização de arquivos de dados de entradas em Excel para serem utilizados nos testes;
- Geração de *scripts* de testes.

O Quadro 2 mostra as tecnologias utilizadas no desenvolvimento da ferramenta.

Quadro 2 - Tecnologias utilizadas na criação da ferramenta

Nome	Descrição
Java	Linguagem de programação, utilizada no desenvolvimento da ferramenta.
Jericho	Manipulação de documentos HTML.
JExcelApi	Manipulação de arquivos em Excel, para gerar os dados de entrada dos testes.
JUnit	Automação dos testes.
Selenium WebDriver	Simulação de testes no navegador *web*.
generatedata.com	Geração de dados aleatórios em Excel para utilização nos testes.

Fonte: Dados do trabalho

Após o desenvolvimento da ferramenta, ela foi utilizada em apoio a alguns testes de sistemas, com o objetivo de detectar erros. Após a detecção de erros, foi realizada uma comparação com o tempo de execução dos testes manuais e também foi verificado se os *scripts* gerados conseguem identificar falhas.

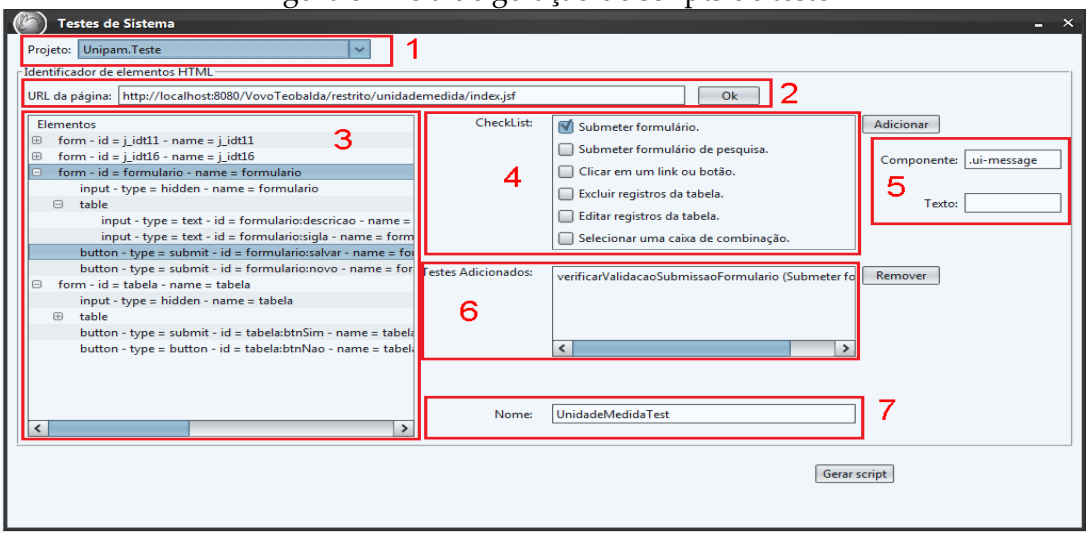
4 DESENVOLVIMENTO E RESULTADOS

Nesta seção será abordada a ferramenta de forma técnica e funcional, para ocorrer o entendimento do código fonte e do seu funcionamento.

4.1 VISÃO FUNCIONAL

A ferramenta de automação de testes realiza a varredura na página *web* que será verificada, possibilitando que o desenvolvedor selecione os elementos HTML que serão utilizados no teste. A Figura 3 apresenta a tela da ferramenta que possibilita a geração dos *scripts* de testes.

Figura 3 - Tela de geração de scripts de teste



Fonte: Dados do trabalho.

Para a geração dos *scripts*, o desenvolvedor utilizará a interface apresentada na Figura 3. A marcação de número 1 representa o projeto ao qual serão gerados os *scripts*. Antes da geração dos testes, o desenvolvedor deve, primeiramente, realizar o cadastro do projeto, o qual possui configurações de diretórios. A marcação de número 2 representa o campo em que será informada a URL da página que será testada, a partir dela a ferramenta carregará alguns componentes HTML.

A marcação de número 3 representa os componentes que foram carregados a partir da página em um formato de árvore. Apenas os elementos importantes para os testes serão disponibilizados para o desenvolvedor selecionar, tais como formulários (<*form*>), tabelas (<*table*>), campos de entrada (<*input*>), caixas de seleção (<*select*>), áreas de texto (<*textarea*>), botões (<*button*>) e *links* (<*a*>). Os testes serão gerados a partir desses componentes.

A marcação de número 4 representa a lista de testes predefinida (*checklist*) disponibilizada pela ferramenta. Ao selecionar um item do *checklist*, o desenvolvedor deverá selecionar alguns elementos da página e/ou até informar algumas informações complementares representadas pela marcação de número 5. O Quadro 3 apresenta os testes definidos pela ferramenta, os elementos e as informação complementares que devem ser informadas a cada tipo de teste gerado.

Quadro 3 - Descrição dos testes definidos pela ferramenta

Teste	Descrição	Componentes HTML que devem ser selecionados	Informações complementares
Submeter formulário.	Permite realizar a submissão de um formulário contendo dados de entrada para inserção de dados.	Formulário; Botão ou link de submissão; Tabela (se for selecionado uma tabela será verificado se ela foi atualizada corretamente, caso a mensagem de retorno seja de sucesso).	Necessário informar qual a identificação do componente que armazena a mensagem de saída.
Submeter formulário de pesquisa.	Permite realizar a submissão de um formulário contendo dados de entrada e que o retorno será uma tabela com os resultados de uma pesquisa.	Formulário; Botão ou link de submissão.	Necessário informar qual a identificação da tabela que será apresentada após a pesquisa.
Clicar em um link ou botão.	Permite realizar um clique em um botão ou link, para verificar se o redirecionamento está correto.	Botão ou link.	Necessário informar qual a identificação do componente e alguma parte do texto presente na nova página que será redirecionada.
Excluir registros da tabela.	Permite realizar o teste de exclusão de registros em uma tabela. Para isso o botão ou link de excluir deve conter o código do elemento que será excluído. Se a mensagem de retorno for de sucesso será verificado se o registro desapareceu da tabela.	Botão ou link de exclusão.	Necessário informar qual a identificação do componente que armazena a mensagem de saída.
Editar registros da tabela.	Permite realizar a edição de um registro. Para isso o botão ou link de editar deve conter o código do elemento que será editado. Após isso será realizado a submissão do formulário, verificado se o item foi alterado.	Formulário; Botão ou link de edição; Tabela (se for selecionado uma tabela será verificado se ela foi atualizada corretamente, caso a mensagem de retorno seja de sucesso).	Necessário informar qual a identificação do componente que armazena a mensagem de saída.
Selecionar uma caixa de combinaçã o.	Permite realizar o evento de alteração de um item em uma caixa de seleção, possibilitando verificar se foi atualizado algum componente a partir desse evento.	Caixa de seleção.	Necessário informar qual a identificação do componente e o texto presente que será carregado após a seleção de um item.

Fonte: Dados do trabalho.

Para garantir a eficácia dos testes, as mensagens do software devem ser padronizadas. Quando uma requisição for realizada com sucesso, deve retornar uma mensagem contendo a palavra "sucesso", se ocorrer qualquer erro inesperado, a mensagem deve iniciar com a palavra seguida por dois pontos "Erro:". Uma validação de dados não é considerada um erro, e sim um tratamento de uma informação de entrada. A partir dessas saídas de mensagens será possível identificar o comportamento de cada teste. No campo componente, deve ser especificado qual elemento da página HTML contém os dados de saída, para verificar se ele está de acordo com o esperado.

A marcação de número 6 representa os testes que foram selecionados para serem gerados. A marcação de número 7 representa o nome do caso de teste que será gerado, esse nome também será utilizado para criar a classe que realizará o teste.

4.2 VISÃO TÉCNICA

Os *scripts* de testes são gerados na linguagem Java, utilizando o *JUnit* e o *Selenium WebDriver*. Mesmo sendo gerados em Java, poderão realizar testes em qualquer sistema, desde que a comunicação com o cliente seja utilizando a tecnologia HTML. O Quadro 4 apresenta o código fonte de um *script* de testes, em que foi selecionado o teste "Submeter formulário".

No método "*init( )*", foi colocada a marcação "*@BeforeClass*" do *JUnit*. Essa anotação irá dizer que toda vez que a classe de teste for executada, primeiramente, deve executar esse método. Dentro dele é informado a URL da página em que o *Selenium* irá executar e também é informado o caminho da planilha que contém os dados de entrada.

O método "*verificarValidacaoSubmissaoFormulario( )*" contém a anotação "*@Test*" do *JUnit*, a qual diz que esse método realiza algum teste. Primeiramente, esse método percorre todas as linhas presentes na planilha de dados, depois são referenciados todos os campos presentes no formulário da página testada e inserido o dado da planilha. Após todos os dados preenchidos, é simulado um clique no botão de salvar e verificado se a saída é a esperada. Caso a saída não seja a esperada, o *JUnit* apresentará que uma falha foi encontrada. Esses campos são recuperados a partir da varredura realizada antes da geração do *script*.

Se, ao gerar o teste, o desenvolvedor selecionar o componente HTML de tabela, a ferramenta irá gerar um código que verifica se a requisição foi um sucesso e percorrerá todas as colunas da tabela, verificando se os campos inseridos estão presentes nela. Caso não estejam, o *JUnit* irá dizer que foi encontrada uma falha.

Por fim, o método "*finish( )*" contém a anotação "*@AfterClass*" que será executada após o término de todos os testes da classe, fechando o navegador de internet por meio do *Selenium*.

Quadro 4 - Código fonte de um *script* de teste gerado pela ferramenta

```
1 public class UnidadeMedidaTest {
2
3       private static ExcelDriver excelDriver;
4       private static WebDriver driver = new FirefoxDriver();
5
6       @BeforeClass
7       public static void init() throws BiffException, IOException{
8
driver.get("http://localhost:8080/VovoTeobalda/restrito/unidademedida/index.jsf");
9               excelDriver = new
ExcelDriver(System.getProperty("user.dir")+"/testes/UnidadeMedidaTest.xls");
10              excelDriver.columnDictionary();
11
12      }
13
14      @Test
15      public void verificarValidacaoSubmissaoFormulario() throws InterruptedException {
16
17              for(int i = 1;i < excelDriver.rowCount();i++)
18              {
19                      WebElement campo1 = driver.findElement(By.id("formulario:descricao"));
20                      campo1.clear();
21
campo1.sendKeys(excelDriver.readCell(excelDriver.getCell("formulario:descricao"), i));
22
23                      WebElement campo2 = driver.findElement(By.name("formulario:sigla"));
24                      campo2.clear();
25
campo2.sendKeys(excelDriver.readCell(excelDriver.getCell("formulario:sigla"), i));
26
27                      WebElement btn = driver.findElement(By.id("formulario:salvar"));
28                      btn.click();
29
30                      Thread.sleep(2000);
31
32                      Assert.assertFalse("Erro ao salvar as informações",
driver.findElement(By.className("ui-growl-title")).getText().startsWith("Erro:"));
33
34                      if(driver.findElement(By.className("ui-growl-
title")).getText().trim().contains("sucesso")){
35
36                              List<WebElement> celulas =
driver.findElements(By.xpath(".//*[@id='tabela:tabelaUnidadeMedida']/table/tbody/tr/td"));
37
38                              boolean encontrou = false;
39                              for (WebElement webElement : celulas) {
40                                      encontrou =
webElement.getText().contains(excelDriver.readCell(excelDriver.getCell("formulario:descrica
```

```
o"), i))
41                                          &&
webElement.getText().contains(excelDriver.readCell(excelDriver.getCell("formulario:sigla"),
i));
42                                  if(encontrou) break;
43                              }
44
45                              Assert.assertTrue("Tabela não foi atualizada após a inserção.",
encontrou);
47                      }
48              }
49      }
50
51      @AfterClass
53      public static void finish() {
54              driver.quit();
54      }
55 }
```

Fonte: Dados do trabalho.

Nesse teste, foram criadas duas variáveis, a "*excelDriver*", referente à biblioteca *JExcelApi*, que foi utilizada para a manipulação dos dados de entrada a partir de uma planilha do Excel, e a "*driver*", referente à biblioteca *Selenium WebDriver* que permitirá a simulação automática do software, utilizando o navegador Firefox.

A Figura 4 apresenta a planilha com os dados de entrada, ela foi gerada utilizando uma ferramenta presente no site http://www.generatedata.com/, que possibilita a geração de dados aleatórios separados por campos.

Figura 4 - Planilha de dados de entrada

	A	B	C
1	**formulario:descricao**	**formulario:sigla**	
2	lobortis ultrices. Vivamus rhoncus. Donec est. Nunc	eleifend. Cras sed leo.	
3	Quisque imperdiet, erat nonummy ultricies ornare, elit	sed dictum eleifend, nunc risus varius orci, in	
4	sed pede nec ante blandit viverra.	neque sed sem egestas blandit. Nam	
5	orci. Donec nibh. Quisque nonummy	nunc ac mattis ornare, lectus	
6	nec tellus.	tristique pellentesque, tellus sem mollis	
7	Fusce fermentum fermentum arcu.	tellus lorem eu metus.	
8	libero at auctor ullamcorper, nisl arcu iaculis enim, sit amet	ipsum ac mi eleifend egestas. Sed pharetra, felis eget varius	
9	at risus. Nunc ac sem ut	Sed neque. Sed eget lacus.	
10	porta elit, a feugiat tellus lorem eu	ipsum primis in faucibus orci luctus et	
11	molestie dapibus ligula. Aliquam erat volutpat. Nulla	vitae aliquam eros turpis non enim. Mauris	
12	at	eu	
13	Donec tempor, est	orci, adipiscing non, luctus sit	
14	nonummy ac, feugiat non, lobortis quis, pede. Suspendisse	amet ante. Vivamus	
15	nunc sed	leo elementum sem, vitae aliquam eros turpis non enim. Mauris	

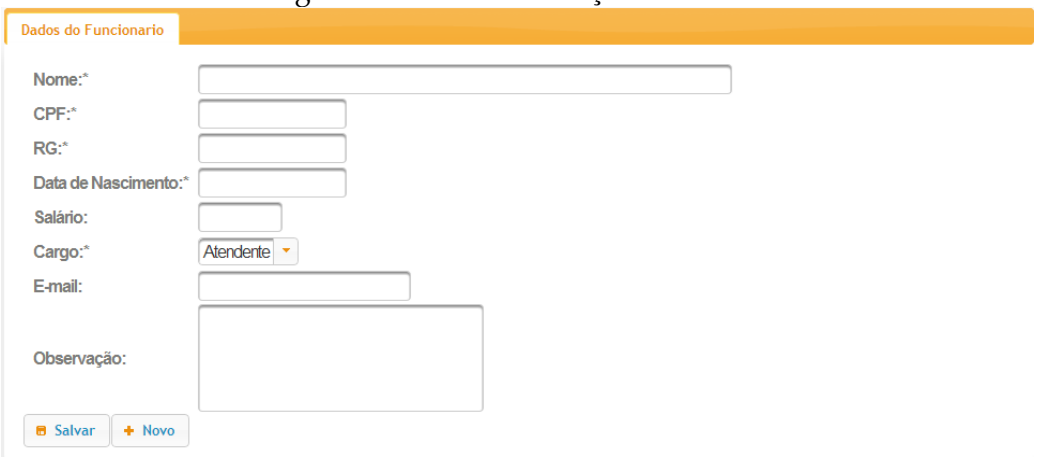
Fonte: Dados do trabalho.

Os dados presentes na coluna A com o cabeçalho denominado "*formulário:descricao*" serão inseridos a partir do *Selenium* como entrada no campo de texto identificado como "*formulário:descricao*", presente na página HTML.

4.3 RESULTADOS

Para verificar a efetividade da geração de *scripts* de testes automatizados, foi realizada uma comparação com os testes manuais em uma interface com oito campos de entrada, como apresenta a Figura 5.

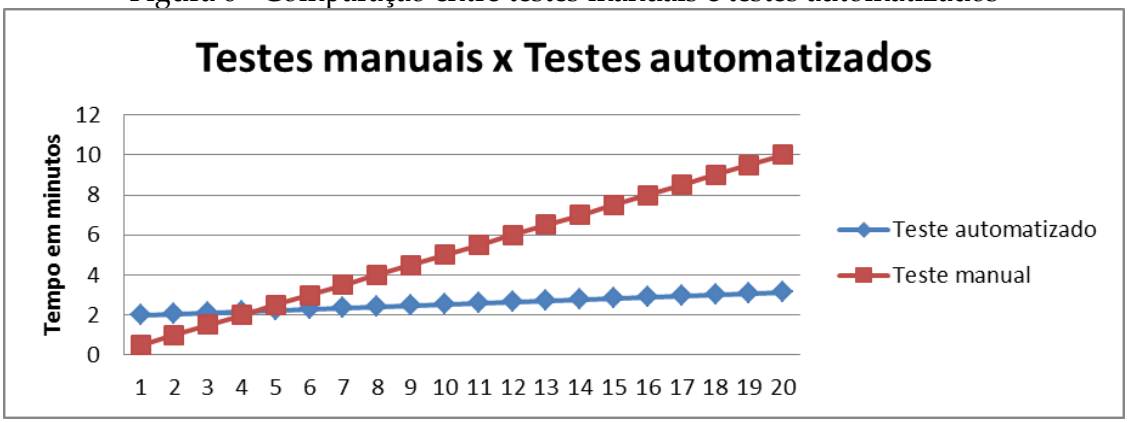
Figura 5 - Tela de realização dos testes



Fonte: Dados do trabalho

Nessa interface, foi testada a inserção de novos registros. Para realizar os testes, foram utilizados cerca de 20 casos de testes, ou seja, o botão salvar foi acionado vinte vezes com entradas diferentes, a fim de identificar possíveis erros. A Figura 6 apresenta um gráfico com a relação do tempo gasto na realização do teste manual e do teste automatizado.

Figura 6 - Comparação entre testes manuais e testes automatizados



Fonte: Dados do trabalho.

No teste manual, foram gastos em torno de trinta segundos em cada caso de teste, foram realizados vinte testes, totalizando cerca de 10 minutos para a realização completa dos testes. No teste automatizado, inicialmente, foram gastos 2 minutos para a geração do *script* utilizando a ferramenta, geração do arquivo que contém os dados

de entrada e na inicialização do teste. Na execução dos testes, foram gastos 3 minutos e 10 segundos para a realização dos vinte testes.

Devido à utilização dos testes automatizados gastarem menos tempo, o desenvolvedor poderá realizar uma quantidade maior de testes para possibilitar uma abrangência maior na detecção de erros. Por exemplo, no teste descrito, poderia ser executado 135 vezes para totalizar 10 minutos de testes, tempo igual se fossem realizados 20 testes manuais. A quantidade de testes a serem realizados dependerá da complexidade dos requisitos.

Outra vantagem do teste automatizado é se houver a necessidade de executá-los novamente. Se o teste for manual, o desenvolvedor deverá realizar todos os testes novamente, gastando mais 10 minutos para a realização dessa tarefa. Por outro lado, o teste automatizado já está definido, basta o desenvolvedor executá-lo novamente, gastando em torno de 1 minuto e 10 segundos, pois não há necessidade de criar o *script* de teste novamente.

Portanto, a realização de testes automatizados com o apoio de ferramentas pode reduzir tempo de desenvolvimento, consequentemente, reduzirá custos e poderá ser mais atuante na detecção de falhas.

5 CONCLUSÃO

O desenvolvimento da ferramenta teve como objetivo criar um software que permitisse a geração de *scripts* de testes, agilizando e melhorando a qualidade dos testes de sistema. A partir da criação dos testes manuais em comparação com a criação dos testes automatizados, foi possível mensurar que a ferramenta proporciona um aumento de produtividade na etapa de testes, consequentemente aumenta as chances da etapa de teste não estourar os custos e cronogramas estabelecidos.

No desenvolvimento da ferramenta, houve algumas dificuldades para estabelecer qual o foco de testes ela implementaria, pois a ferramenta teria que realizar testes em softwares desenvolvidos em diferentes linguagens de programação desde que a comunicação com o usuário utilizasse a tecnologia HTML. Ainda houve dificuldade em criar um padrão de *script* que rodasse automaticamente, simulando a execução do software real, e identificasse as saídas para detectar os possíveis erros.

Portanto, a ferramenta auxilia os desenvolvedores que desejam realizar testes de sistema a partir da interface *web*, de acordo com a lista de testes disponibilizada pela ferramenta, permitindo que eles agilizem as tarefas de testes mais simples, focando nos testes mais complexos e específicos.

Um teste efetivo é aquele que detecta o maior número de falhas em um software. Utilizando as técnicas de detecção de erros de caixa preta e caixa branca em conjuntos, possibilita que os testes sejam mais abrangentes, provavelmente conseguindo detectar mais erros em um software. Portanto, futuramente, pretende-se desenvolver na ferramenta um módulo para detecção de erros a partir dos testes de unidade e um módulo para geração de dados de entrada automaticamente a partir dos campos do formulário a ser testado. Assim, a ferramenta poderá atuar em dois vértices para capturar erros e garantir o menor número de falhas possíveis, colaborando para a qualidade do produto final.

REFERÊNCIAS

BERNARDO, Paulo Cheque. *Padrões de testes automatizados***.** 197 f. Dissertação (Mestrado) - Departamento de Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2011.

BURNS, David. *Selenium 2 Testing Tools:* Beginner's Guide. Birmingham: Packt Publishing, 2012.

CRAIG, R.D., JASKIEL, S. P. *Systematic Software Testing.* Boston: Artech House Publishers, 2002.

GUERRA, Ana Cervigni; COLOMBO, Regina Maria Thienne. *Tecnologia da Informação:* qualidade de produto de software. Brasilia: PBQP, 2009. 429p.

HUMPHREY, W. S. *A Discipline for Software Engineering.* Addison Wesley, 1995.

LUCINDA, Marco Antônio. *Qualidade:* fundamentos e práticas. Rio de Janeiro: Brasport, 2010.

MAGNANI, G. *Melhoria de processo:* visão geral e estudos de caso. Tutorial oferecido na Semana de Engenharia de Software, 3, 12 ago. 1998, São Paulo.

MASSOL, Vincent; HUSTED, Ted. *JUnit in Action.* Greenwich: Manning Publications, 2009.

OAKLAND, John S. *Gerenciamento da qualidade total.* São Paulo: Nobel, 1994.

PRESSMAN, Roger. *Engenharia de software.* 3. ed. São Paulo: Pearson, 1995.

ROCHA, Ana Regina Cavalcanti da; MALDONADO, José Carlos; WEBER, Kival Chaves. *Qualidade de software:* teoria e prática. São Paulo: Prentice Hall, 2001.

SOMMERVILLE, Ian**.** *Engenharia de Software.* 9. ed. São Paulo: Pearson, 2011.